Two Dimensional Visualization of Software Metrics

TIBOR BRUNNER and ZOLTÁN PORKOLÁB, Ericsson Ltd.

Successful software systems are under continuous change. Bug-fixing, adding new features modify or extend additional code to the existing code base. Under these activities we must be aware of software quality, avoid its degradation and alert when a major code refactoring seems to be inevitable. For this purpose we must continuously collect quality related data from the software: static analysis results, software metrics and other statistics. However, data have to be analyzed and presented in a way that the architects and designers could comprehend the information in various context: e.g. related to the number of changes on the code, with the relative distribution of the issues and in connection with the complexity of the given module.In this paper we show how to collect some of these key quality indicators and how to present them in a clear manner so architects and developers can overview the quality factors organized by nested components of the program in a multidimensional way. This method allows programmers to reason about correctness of the architecture, identify critical components and decide about necessary actions, like refactoring the architecture.

Categories and Subject Descriptors: D.2.8 [**Metrics**]: Complexity measures—*Visualization*; H.1.2 [**Models and Principles**]: User/Machine Systems—*Human Information Processing*

General Terms: Software metrics, Human Factors

Additional Key Words and Phrases: Software complexity, Two dimensional, Visualization

1. INTRODUCTION

Static program analysis is a methodology which means the observation of a software without its execution. Static analysis techniques target testing, correctness checking, comprehending and other purposes [Crawford et al. 1985; Gyimothy et al. 2005]. CodeCompass [Ericsson 2017] is a static analysis framework which gives opportunity to perform various kind of analysis on the source code and present the results in different visualization methods. The architecture of this framework consists of two orthogonal layers. On a vertical layer it has a classical server-client architecture, since the statically collected information is presented by a web server towards a GUI or other querying client scripts. And on the horizontal layer the server and the client are implemented as independent plug-ins.

Plug-ins provide the different functionalities of the framework. These can examine the code base from a wide range of aspects: some of them are language parsers which collect the named entities of a given language (variables, functions, classes, etc.) for further processing, others are inspecting the version control history of the project, yet others provide additional data from external sources, like third party databases.

In this paper first in Section 2 we overview the main features of CodeCompass, an open source code comprehension framework. We discuss the most important software metrics we measure in Section 3.

Author's address: Tibor Brunner, Eötvös Loránd University, Pázmány Péter sétány 1/C, Budapest, Hungary, H-1117; email: bruntib@caesar.elte.hu; Zoltán Porkoláb, Eötvös Loránd University, Pázmány Péter sétány 1/C, Budapest, Hungary, H-1117; email: gsd@caesar.elte.hu

In: Z. Budimac (ed.): Proceedings of the SQAMIA 2017: 6th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Belgrade, Serbia, 11-13.9.2017, Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

To collect measurement data we use plug-ins for CodeCompass. We introduce them in Section 4. Our paper concludes in Section 5.

2. CODECOMPASS

Bugfixing or new feature development requires a confident understanding of all details and consequences of the planned changes. For long existing stems, where the code base have been developed and maintained for decades by fluctuating teams, original intentions are lost, the documentation is untrustworthy or missing, the only reliable information is the code itself. Code comprehension of such large software systems is an essential, but usually very challenging task. As the method of comprehension is fundamentally different from writing new code, development tools are not performing well.

During the years, different programs have been developed with various complexity and feature set for code comprehension but none of them fulfilled all requirements. CodeCompass [Ericsson 2017] is an open source framework developed by Ericsson Ltd. and the Eötvös Loránd University, Budapest to help understanding large legacy software systems. Based on the LLVM/Clang compiler infrastructure, CodeCompass gives exact information on complex C/C++ language elements like overloading, inheritance, the (read or write) usage of variables, possible calls on function pointers and the virtual functions – features that various existing tools support only partially.The wide range of interactive visualizations extends further than the usual class and function call diagrams; architectural, component and interface diagrams are among the few of the implemented graphs.

To make comprehension more extensive, CodeCompass is not restricted to the source code. It also utilizes build information to explore the system architecture as well as version control information when available: git commit history and blame view are also visualized. Clang Static Analyzer results are also integrated to CodeCompass. Although the tool focuses mainly on C and C++, it also supports Java and Python languages.

A plug-in can introduce a model schema. CodeCompass establishes a connection to a relational database system and provides an Object Relational Mapping (ORM) tool to handle the persistence of ordinary C++ objects. Besides relational databases the given plug-in can also store its data in an arbitrary alternative database system or even in a single text file. The plug-in can provide a parser which fills this database. Currently CodeCompass contains many different type of parsers. Some of them parse the source code of different programming languages, others gather source control information from the Git database, or do text-search indexing. In this paper we will discuss the *Metrics* plug-in which provides metrics-related data in details in Section 3. The dataset collected by the parser is transfered to the client by a web server. This server listens on a port and routes the client queries based on the given URL to the single plug-ins. There are well defined interfaces between the client and the server, or the plug-in itself can also define its own API, since as an independent module of the framework only the plug-in has information about the structure of the stored dataset. The API can be accessed by a client program through a Remote Procedure Call (RPC) methodology. CodeCompass provides a web based GUI which enables different visualizations of the provided information. The GUI is also expansible by the plug-ins using JavaScript modules.

Having a web-based, pluginable, extensible architecture, the CodeCompass framework can be an open platform to further code comprehension, static analysis and software metrics efforts. CodeCompass also a good starting point to develop a Language Server Protocol Clang Daemon [Microsoft 2017] prototype.

3. METRICS

Metrics are some quantitative measurements of a software. These are aiming to describe a project based on a given perspective [Fenton 1991].

3.1 Lines of Code

Maybe the most important metric is the *Lines of Code*. This is a simple way of describing the size of a project. There are subversions of this metric which give a more subtle picture: we can count the number lines which contain only comments, we can omit blank lines and we can compute pure source lines separately. This metric is independent from the programming language it is applied on.

3.2 Cyclomatic complexity

A more sophisticated way of measuring complexity of the software is *Cyclomatic* or *McCabe* metric [McCabe 1976]. This measures the linearly independent paths in a control flow. This metric can be easily applied on procedural languages, since it is determined by the number of decisions in the program. Note that this is also the number of parts the planar graph of the control-flow diagram divides the surface. Unfortunately this metric does not make distinction between the nested and sequential branches, though intuitively nested loops or conditionals are considered more complex than sequential ones.

3.3 Tight coupling

There are metrics which reflect on the architecture of the program on a higher level. Different types of *Coupling* can be defined which indicate how much the modules are independent from each other [Chidamber and Kemerer 1994; Henderson-Sellers 1996]. E.g. *data coupling* is tight when much information is shared between the modules via procedure parameters or global variables, or *control coupling* is tight when the module is controlling the flow of another by instructing it what to do, etc.

3.4 Runtime metrics

Not only static time metrics can be defined but also runtime ones which can be collected during program execution. These are like *execution time* and *load time* measurements or *coverage* metrics which show the ratio of the source code covered by a test suite and the amount of uncovered parts.

3.5 Number of bugs metrics

CodeCompass supports the compile time analysis of the project. This way the before-mentioned metrics can be collected. But it can also invoke external tools to gather quantitative descriptors of modules. LLVM/Clang is a compiler infrastructure which parses source codes in C++ language, builds its Abstract Syntax Tree (AST) which contains semantic information too. This enables programmers to run several checks on the program. *Static Analyzer* and *Clang Tidy* are two tools inside Clang which discover typical programming issues and misuses of the language. Some of these checks are simple enough so only consideration of the AST is sufficient: `using namespace` is strongly contraindicated in a header file. Its existence can be easily checked by inspecting the AST. Some others need path sensitive examination of the source: we can find division by zero or null pointer dereferences by following the control flow and note the possible values of the divisor or the pointer respectively. The technique used here is called symbolic execution which means the interpretation of the source code. Note that this is still a static analysis technique despite it has information about the symbolic values of variables and more complex expressions.

CodeChecker aims to collect the bug findings of these Clang tools and to store them in a database. The bugs can be queried through a public API. This way CodeCompass can access the number of errors committed in the source code. This gives a metric which describes the quality of a module.

4. METRICS PLUG-IN IN CODECOMPASS

In this section we describe the Metrics plug-in of CodeCompass from the database layout and the parser through the service to the GUI.

4.1 Database layout

Originally CodeCompass was developed as a code comprehension tool for large code bases. Scalability was always an important aspect since large-scale projects are not rare in industrial environment. This is the reason why not the whole AST is stored in the database. It turned out that for most tasks in code comprehension storing the named entities (classes, functions, variables, etc.) is sufficient. These are enough for navigating through the source code and inspect its regions.

The metrics differ from this in the sense that not only named language elements may possess metrics but files too. For example we can take the *Lines of Code* metrics for functions, classes and for files as well. In CodeCompass we chose a visualization method which works on the granularity of directories and source files. Thus in the database the following fields take place:

id. Unique identifier of the metric.

file. Identifier of the file to which the metric belongs.

metric. Unsigned value of the metric.

type. The type of the metric, like McCabe, Lines of Code, Number of Bugs, etc.

4.2 Parser

Usually it is the parser's job to collect all the information which is presented by the web service. However practically sometimes the service has enough information to answer some queries by computing the result on-the-fly. We decided that Lines of Code and McCabe metrics are computed while parsing the project. One reason is that the amount of information to store is proportional to the number of files which requires much less space compared to the other tables of the database (e.g. the AST node descriptor tables). The other reason is that some metrics are dependent on the programming language. Although it would be a good approximation of McCabe metrics to count the number of `if`, `while`, `for`, etc. keywords in a source file which introduce a control structure, but the answer would not be precise in case of comments or string literals containing these keywords. It is more convenient to use a language parser to build the AST and to count the control structures in it. However the disadvantage of this solution is that technically we loose the language independence, since building the AST requires a language parser added as a plug-in in CodeCompass.

4.3 Service

The main role of service layer is to serve the client. The question is that how much logic should the service perform; most of the times simply reading the database is sufficient but sometimes it is more convenient to accomplish a quick computation of which the result is not worth to store at parsing time. Another case when it is advantageous to entrust the service by doing the job is when the information is stored in a separate database. This is the case at CodeChecker where the checker results, i.e. the specific bugs of the source code are stored in an external database which can be accessed via another web interface. In CodeCompass we give the opportunity to visualize as many kind of metrics as many can be collected either in parsing time or in the service. Thus the service of Metrics plug-in provides one API function for gathering a specific metric for a given file: `int getMetrics(FileId, Metric type)`.

4.4 Graphical User Interface

CodeCompass provides a web-based user interface which can be opened in a browser. This enables us to create any spectacular visualization using the modern JavaScript frameworks and libraries. This also helps to achieve the largest user base, since no thick client application is required for browsing analysis results. In the literature we can find a big variety of visualization methods [Langelier et al. 2005]. Many times these methods are aiming to represent several dimensions in one picture which requires independent views. For this purpose the 3D space can be used [Wettel and Lanza 2008], or the texture of the diagram elements [Holten et al. 2005]. A common visualization for software metrics is the Treemap which means separate regions filling a surface and indicating the quantity by the area of a region [Balzer et al. 2005]. As for the metrics' visualization we chose such a two-dimensional Treemap representation. In this view the source code hierarchy is visualized as rectangular boxes. Each box belongs to a directory or a file in the current folder. Those which belong to a directory can be clicked which event triggers a zooming animation which leads into the content of that directory. This view is two-dimensional, because the different metrics can be assigned either to the size of a box or to its color. For instance it can be set that the more lines of code is contained by a file, the bigger its size is. Or the more bugs are found in a specific directory, the bluer its color is.

In the picture below we can see an example of metrics' usage. In this example the CodeCompass source code itself is parsed and analyzed. The used metrics are the *lines of code* in size dimension and the *number of bugs* in color dimension. In the first image we can see the content of the *service* directory. The largest subdirectory with LOC metrics is *core-api* since it has the biggest rectangle in the top left corner. The second largest is *language-api*, etc. In *codechecker-api* folder we can see a blue rectangle which indicates that it has a subdirectory with at least one found bug. By clicking on this rectangle we get the second picture which navigates towards *include* directory where the actual file (*codeCheckerDBAccess.h*) with a found bug can be seen.

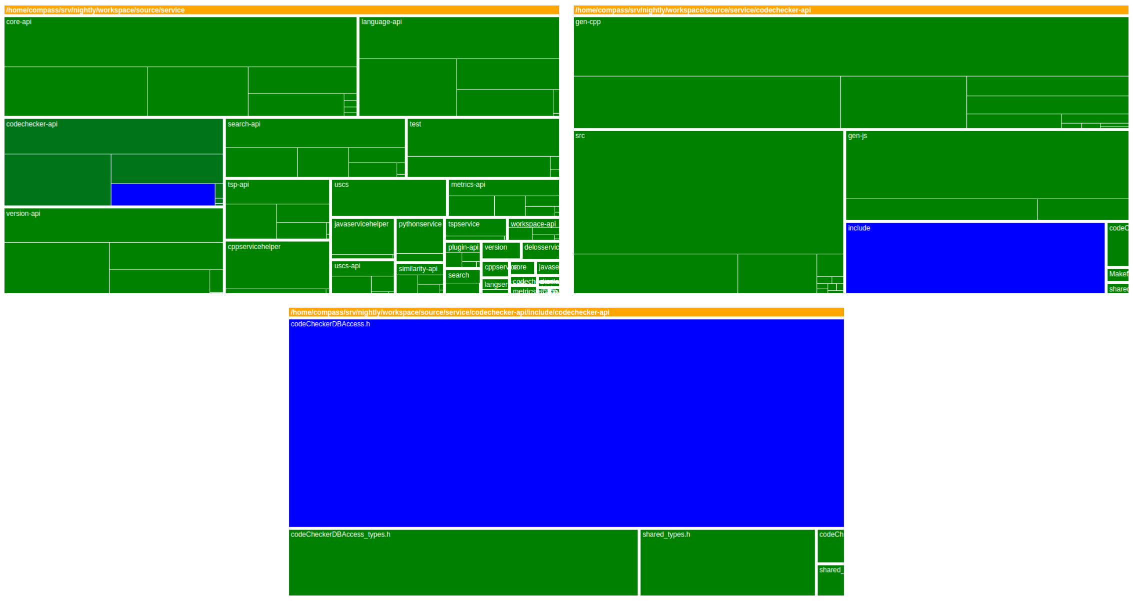

Fig. 1. Screenshot of metrics view in CodeCompass.

5. CONCLUSION

In this paper we discussed a new approach to collect and visualize metrics data for large scale software systems. Our solution is implemented as a plug-in of the CodeCompass open source software comprehension platform. The parser plug-in collects the required data into a database, and the CodeCompass server provides data for the clients as a service. The web-based graphical user interface is aimed to show various two-dimensional visualizations to reviel connections between the various components of the software system.

REFERENCES

Michael Balzer, Oliver Deussen, and Claus Lewerentz. 2005. Voronoi treemaps for the visualization of software metrics. *SoftVis '05 Proceedings of the 2005 ACM symposium on Software visualization* (2005), 165–172.

Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.

S.G. Crawford, A.A. McIntosh, and D. Pregibon. 1985. An analysis of static metrics and faults in C software. *Journal of Systems and Software* 5, 1 (1985), 37 – 48. DOI:http://dx.doi.org/10.1016/0164-1212(85)90005-6

Ericsson. 2017. CodeCompass: a code comprehension framework. (2017). Retrieved June 1, 2017 from https://github.com/Ericsson/CodeCompass

Norman E Fenton. 1991. Software metrics: a rigorous approach. (1991).

Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. 2005. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software engineering* 31, 10 (2005), 897–910.

Brian Henderson-Sellers. 1996. Object-Oriented Metrics. Measures of Complexity. (1996).

D. Holten, R. Vliegen, and J.J. van Wijk. 2005. Visual Realism for the Visualization of Software Metrics. *Visualizing Software for Understanding and Analysis, 2005. VISSOFT 2005. 3rd IEEE International Workshop* (2005).

Guillaume Langelier, Houari Sahraoui, and Pierre Poulin. 2005. Visualization-based Analysis of Quality for Large-scale Software Systems. *ASE '05 Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering* (2005), 214–223.

J. McCabe, Thomas. 1976. A Complexity Measure. *IEEE Transactions on Software engineering* 2, 4 (1976), 308–320.

Microsoft. 2017. Language Server Protocol. (2017). Retrieved June 1, 2017 from https://github.com/Microsoft/language-server-protocol

Richard Wettel and Michele Lanza. 2008. CodeCity: 3D visualization of large-scale software. *ICSE Companion '08 Companion of the 30th international conference on Software engineering* (2008), 921–922.